JFlow: Practical Refactorings for Flow-based Parallelism

Nicholas Chen, Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
{nchen, rjohnson}@illinois.edu

***Abstract*—Emerging applications in the domains of recognition, mining and synthesis (RMS); image and video processing; data warehousing; and automatic financial trading admit a particular style of parallelism termed *flow-based parallelism*. To help developers exploit flow-based parallelism, popular parallel libraries such as Groovy's GPars, Intel's TBB Flow Graph and Microsoft's TPL Dataflow have begun introducing many new and useful constructs. However, to reap the benefits of such constructs, developers must first use them. This involves refactoring their existing sequential code to incorporate these constructs – a manual process that overwhelms even experts. To alleviate this burden, we introduce a set of novel *analyses* and *transformations* targeting flow-based parallelism. We implemented these ideas in JFlow, an interactive refactoring tool integrated into the Eclipse IDE. We used JFlow to parallelize seven applications: four from a previously known benchmark and three from a suite of large open source projects. Our evaluation on these applications demonstrates that JFlow, with some minimal interaction from the developer, can successfully parallelize applications from the aforementioned domains with good performance (offering up to 3.45x speedup on a 4-core machine) and is fast enough to be used interactively as part of a developer's workflow.**

I. Introduction

The term "flow-based parallelism" is coined after Morrison's *Flow-Based Programming: A New Approach to Application Development* [1]. Flow-based parallelism is a parallel programming paradigm that partitions an expensive computation into a directed graph, i.e., a computation flow graph. Each node in the graph embodies part of the original computation and the edges between nodes represent dependencies. *Data flow between nodes*. Nodes perform their computation when all their data dependencies are available. Parallelism is achieved when nodes can operate concurrently. Examples of flow-based parallelism include pipeline parallelism, event-based coordination and the wavefront pattern [2], [3]. The structure of the graph is usually simple, e.g., in the case of pipeline parallelism but could also be more complex, e.g., in the case of wavefront computations.

Flow-based parallelism offers many advantages. It is well-suited for parallelizing emerging applications in the domains of recognition, mining and synthesis (RMS); image and video processing; data warehousing; and automatic financial trading (Section IV). Using flow-based parallelism not only improves the performance of such applications but also their modularity. By partitioning the computation into nodes of a graph that communicate via explicit parameters, we can reduce the coupling between different parts of the software, making it easier to maintain and evolve each part independently.

A compelling way to leverage flow-based parallelism is to refactor existing sequential code to use the constructs of a parallel library such as Groovy's GPars [4], Intel's TBB Flow Graph [5] or Microsoft's TPL Dataflow [6]. Our prior work examined the constructs in an older version of Intel's TBB and found them expressive enough to address many of the common idioms found in applications [7]. We found using such constructs to be more advantageous compared to direct parallelizing with low-level threads. Firstly, programs written using parallel constructs were more succinct and easier to evolve because the library provides useful constructs for adding/removing nodes or edges to a computation graph. Secondly, in some cases, code parallelized using library constructs were faster than code parallelized directly with low-level threads because the library has better support for managing and coordinating threads through a thread pool.

To refactor their code to use such constructs, developers face **two** overwhelming tasks: *analysis* and *transformation* of their sequential code. First, a developer has to partition the original sequential code into a computation flow graph with nodes and edges. She must scrutinize each node to ensure that no data races exist. Given a large and unfamiliar code base, this is hard even for experts. Second, once she is convinced that there are no data races, she still has to meticulously write repetitive, boilerplate code to create the nodes and link the edges between nodes. Given a large computation graph, there could be many nodes and edges, and manually writing code to describe them is tedious. *How can we assist developers in analyzing and transforming their sequential programs?*

Parallelizing compilers attempt to relieve the burden of analyzing and transforming the code in an entirely *automated* manner, without involving the developer. While appealing, even the most advanced static analysis in the compiler are frequently confounded by common programming idioms that abound in typical object-oriented programs, forcing the compiler to be overly pessimistic and precluding many parallelization opportunities. Kim et al. [8] tested two commercial C/C++ compilers on the OmpSCR benchmarks [9] and found that, at best, the compilers could only automatically parallelize 4 out of 13 loops although loop parallelism had been studied since the first parallelizing compilers [10]. They attributed the reasons for failure to pointer based accesses, complex

control flow and insufficient cost-benefit analysis. Even state of the art static analysis for object-oriented program [11]–[13] have trouble reasoning about many best practice idioms. For instance, in the applications we evaluated, we found that the use of static factory methods, logging constructs and, in general, the use of fairly standard library APIs confuse the static analysis and force it to be safe but overly conservative. Common techniques for increasing the precision of static analysis (at the expense of memory and running time) e.g., increasing k for k-object sensitivity [14] or n for call-string-sensitivity [15], do not help.

While it is difficult for static analysis to reason about such idioms, it is easier for a human. Thus, we propose a practical approach that combines static analysis with human interaction. Our approach actively engages the developer, i.e., the domain expert in the parallelization process. We adapt ideas from human automation and divide the tasks between developer and tool such that each performs what each excels at [16]. The developer performs the *high-level* reasoning task that tools have problems with — identifying which parts of the original computation to partition — whereas the tool handles the *low-level* analysis task that developers find tedious — analyzing if the partitions have data races and generating the parallel code. This combination is effective. It enables us to parallelize many more applications than would have been possible through static analysis alone.

We implemented this interactive approach in our tool, JFlow. JFlow statically analyzes the code and reports back to the developer. As a tool, JFlow has high *utility* and is useful in many scenarios. When the analyses are successful, the tool performs the transformation, generating source code that targets the constructs of a parallel library. Even when the analyses fail, the tool is useful — it pinpoints and reports the problems. This allows the developer to either fix the problems and retry, or proceed with the transformation directly if she deems the reported problems innocuous (e.g., as determined by her suite of test cases). JFlow can even be used exploratorily: if the developer wishes to inspect for data races, she could use the tool to analyze the code, view the results and forego the transformation step. She could even use JFlow as a code generator, ignoring all analysis results and merely using it to generate code that she will later tweak by hand. Our tool operates at the source code level. Thus, all issues reported can be easily inspected and understood by the developer. Additionally, the transformations can be inspected and tweaked by the developer, as desired.

Our contributions are twofold. First, we introduce our analyses and transformations that specifically target flow-based parallelism. In this paper, we target the most common flow-based parallelism, i.e., pipeline parallelism, that we have observed in our suite of applications. The heart of our analyses is a novel approach for detecting *inter* and *intra* node data-races. Our approach combines k-object sensitivity with ownership transfer inference — an approach that works particularly well for flow-based parallelism with their regular flows of coarse-grained data (see Section III). Our transformations generate human-readable source code to describe the computation flow graph. The ideas behind our analyses and transformations are *general* and can be adapted for different languages and frameworks. Second, we incorporated our analyses and transformations into our interactive tool, JFlow. JFlow works with Java applications and targets the constructs from Groovy's GPars parallel library. We evaluated JFlow on seven applications from different emerging domains. We report the parallel speedups and discuss the user interactions necessary for each application.

The source code for JFlow and the applications that we evaluated are available at http://vazexqi.github.io/JFlow/.

II. Motivating Example

Consider LIRe (16K SLOC), an open source Java content-based image retrieval (CBIR) library [17]. LIRe takes a query image, extracts features from it and, based on those features, retrieves similar images from a database of candidate images. Typical features that are extracted using computer vision techniques include color histograms, shapes and textures.

In LIRe, the main bottlenecks to performance are the indexing and retrieval stages. This example focuses on the indexing stage. In the indexing stage, a set of images is analyzed to build the database of candidate images. During the analysis, features are extracted from the images and stored in the database. Because there isn't a single feature that works across all images, multiple features are usually extracted and stored in the database. The bottleneck in the indexing stage comes from the number of features extracted and the number of images that need to be analyzed (typically in the hundreds of thousands for a decent size database).

Figure 1 shows the indexing loop for LIRe with four feature extractors. Consider a developer who decides to parallelize this loop using flow-based parallelism. She decides to partition each feature extractor into its own node. The flow graph for such a partition, i.e., a pipeline, is shown on the right hand side of the figure. *How could she make use of JFlow to parallelize her code?* **Three** steps: annotate, extract nodes and invert loop.

First, she would need to annotate her desired partitioning; we use simple comments for now. While there have been prior work [18], [19] on using the *program dependence graph* [20] to perform automatic partition, our own experience shows that it does not work as well for the kinds of modern object-oriented programs that we are interested in. Complex heap data dependencies in the program dependence graph lead to very fine-grained partitions that have high communication costs when parallelized. We found it more effective to rely on the developer to provide a suitable partition as a starting point, as done by Thies et al. [21] and Jenista et al. [22].

After annotating her code, the developer invokes JFlow. JFlow currently supports two refactorings. The ideas for both refactorings, Extract Node and Invert Loop, were conceived based on our past experiences parallelizing applications using parallel library constructs [7].

The first refactoring that JFlow performs is Extract Nodes. This refactoring attempts to create nodes from the developer's annotations and links each node with other nodes

```
1  for (String imagePath : FileUtils.getAllImages(new File(IMAGES_DIRECTORY))) {
2
3      // Begin Node1
4      BufferedImage bufferedImage= ImageIO.read(new FileInputStream(imagePath));
5      Document docJPEG= JPEGExtractor.createDocument(bufferedImage, imagePath);
6      // End Node1
7
8      // Begin Node2
9      Document docTamura= tamuraExtractor.createDocument(docJPEG, bufferedImage, imagePath);
10     // End Node2
11
12     // Begin Node3
13     Document docColor= autoColorCorrelogramExtractor.createDocument(docTamura, bufferedImage, imagePath);
14     // End Node3
15
16     // Begin Node4
17     Document docFCTH= FCTHExtractor.createDocument(docColor, bufferedImage, imagePath);
18     indexWriter.addDocument(docFCTH);
19     // End Node4
20 }
```

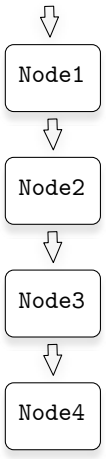

Fig. 1. The indexing loop for LIRe with the different nodes annotated.

through their data dependencies. In its analysis step, it statically checks for *inter*-node data races (i.e., data races *between* nodes) and warns the developer of any. In its transformation step, it creates the nodes and the edges between them using the underlying library constructs.

The underlying construct to represent a node is a *closure*. The body of the closure contains the annotated statements for each node. Closures as a built-in language construct are supported in C++11 and C#. In the current version of Java, closures are expressed using anonymous inner classes. The underlying construct to represent an edge is a *channel*. A channel is a strongly-typed queue data structure. All three parallel libraries, i.e., GPars, TBB and TPL Dataflow, follow this same scheme of using closures and channels. Thus, our approach is very applicable to all three and we can easily target the constructs in each of them.

Figure 2 shows the generated code for `Node2` after the EXTRACT NODES refactoring. `Node2` is expressed as a `DataflowMessagingRunnable` anonymous inner class (lines 15 – 26). It *consumes* values from `channel2` (line 26) and *produces* values into `channel3` (line 24). The values in `channel2` are provided by `Node1`. The channels are expressed as a `DataflowQueue`. Typically, each data dependency is communicated via one dedicated channel. However, this leads to a proliferation of channels — making the code hard to read — and also comes with an associated performance cost — additional bookkeeping for each channel. We address both these issues by *bundling* data dependencies together when possible (lines 1 – 5).

Observe that there is a non-trivial amount of boilerplate code that needs to be written. Imagine how tedious it would be for a developer to do this by hand. While some amount of verbosity stems from the lack of a direct syntax for closures in Java, the rest are actually required as part of using the underlying parallel library. Note that the generated code is **not** parallel yet. EXTRACT NODES is an intermediate refactoring step. We offer the developer the opportunity to explore the code and make simple tweaks as desired. For instance, perhaps she would like to make use of other refactorings such as RENAME VARIABLE to rename the generic channels names from `channel0` to something more intention-revealing.

The second refactoring that JFlow supports is INVERT LOOP. INVERT LOOP locates all the `DataflowMessagingRunnable` objects in the loop and moves them into a computation flow graph. Intuitively, it inverts the nodes in the loop into a new construct, a `FlowGraph` object, allowing the nodes to run in parallel. Internally, a `FlowGraph` object maintains its own thread pool and coordinates the execution of node.

Figure 3 shows the code after the INVERT LOOP refactoring. Line 1 creates a new `FlowGraph` object, `fGraph`. Lines 3 – 14 register the `DataflowMessagingRunnable` for `Node2` with the `FlowGraph` object.

For each node, JFlow checks if it is possible to run the node in a data parallel manner without any *intra*-node data races (i.e., data races *within* nodes). If so, it *informs* the user of this possibility. The user then needs to *decide* if this is suitable for her particular application. Operating in a data parallel manner improves throughput but does **not** preserve the ordering of processed items. For instance, in the LIRe example, the images will arrive out-of-order if they are processed in a data parallel manner.[1] Allowing this is a decision that the developer needs to make. It is impossible to statically infer this from the sequential code alone. The sequential code is *over-constrained* and always imposes an ordering even when one is not necessary [23]. If we strictly enforce the sequential ordering, we miss many parallelization opportunities.

In the LIRe example, JFlow informs the user that `Node1`, `Node2` and `Node3` can operate in a data parallel manner.

[1]TBB supports in-order processing by *tagging* items. Each item is tagged with its arrival order. The receiving node keeps track of the order and buffers items until it receives an item with the right order. This is convenient but incurs some overhead.

```
1  class Bundle {
2      BufferedImage bufferedImage;
3      Document docColor, docJPEG, docTamura;
4      String imagePath;
5  }
6
7  final DataflowQueue<Bundle> channel0= new DataflowQueue<Bundle>();
8  final DataflowQueue<Bundle> channel1= new DataflowQueue<Bundle>();
9  ...
10 for (String imagePath : FileUtils.getAllImages(new File(IMAGES_DIRECTORY))) {
11     Bundle b= new Bundle();
12     b.imagePath= imagePath;
13     channel0.bind(b);
14     ...
15     new DataflowMessagingRunnable() {
16         @Override
17         protected void doRun(Object... args) {
18             Bundle b= ((Bundle)args[0]);
19             BufferedImage bufferedImage= b.bufferedImage;
20             String imagePath= b.imagePath;
21             Document docTamura= b.docTamura;
22             Document docColor= autoColorCorrelogramExtractor.createDocument(docTamura, bufferedImage, imagePath);
23             b.docColor= docColor;
24             channel3.bind(b);
25         }
26     }.call(channel2.getVal());
27     ...
28 }
```

Fig. 2. The closure representing `Node2` and the channels connecting it.

```
1  FlowGraph fGraph= new FlowGraph();
2  ...
3  fGraph.operator(Arrays.asList(channel1), Arrays.asList(channel2), 4, new DataflowMessagingRunnable() {
4      @Override
5      protected void doRun(Object... args) {
6          Bundle b= ((Bundle)args[0]);
7          BufferedImage bufferedImage= b.bufferedImage;
8          Document docJPEG= b.docJPEG;
9          String imagePath= b.imagePath;
10         Document docTamura= tamuraExtractor.createDocument(docJPEG, bufferedImage, imagePath);
11         b.docTamura= docTamura;
12         bindOutput(b);
13     }
14 });
15 ...
16 for (String imagePath : FileUtils.getAllImages(new File(IMAGES_DIRECTORY))) {
17     Bundle b= new Bundle();
18     b.imagePath= imagePath;
19     channel0.bind(b);
20 }
21 fGraph.waitForAll(); // Wait for all computation in the FlowGraph to complete and then continue execution
```

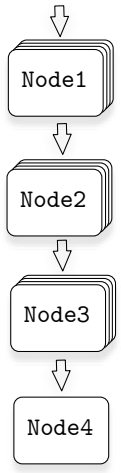

Fig. 3. Inverting the loop so that each node can operate in parallel. Additionally, `Node1`, `Node2` and `Node3` can operate in a data parallel manner.

She decides that it is permissible for items to be processed out-of-order. The third parameter to `operator` on line 3, represents the degree of parallelism to use. By default, JFlow uses the number of cores on the machine. The user can tune this by hand using an external profiler, or an auto-tuning technique [24].

The original loop serves as a *generator* that will provide the initial data to `Node1`. In the figure, the `for` loop will *generate* the `imagePath` variable that is used in the first and subsequent nodes. After the `FlowGraph` object finishes its execution, it resumes serial execution with the rest of the code (Line 21).

III. Analyses and Transformations

This section details the analyses we perform to determine if the Extract Nodes and Invert Loop refactorings can be performed safely. The analyses use the WALA [25] library for analyzing Java source code. Both refactorings are integrated into the Eclipse IDE.

A. Extract Nodes

The analysis for the Extract Nodes refactoring checks if the annotated nodes can be safely partitioned into a pipeline, i.e., no abnormal control flow, no loop-carried dependencies and no inter-node data races. Figure 4 presents an overview of the analysis. We detail each stage below. Earlier stages are less expensive to run; we use their results to determine if we need to run the later, more expensive stages or to abort the analysis.

The developer annotates the original program and partitions the statements in the original loop into N nodes. The original loop iterates over all the elements to be processed and *generates* the data for the first node in the pipeline.

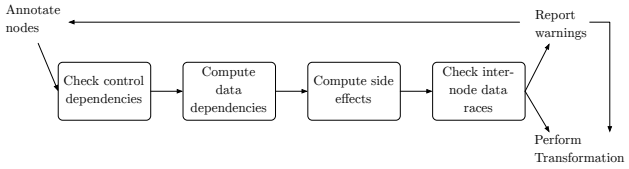

Fig. 4. Overview of the analysis workflow for the EXTRACT NODES refactoring.

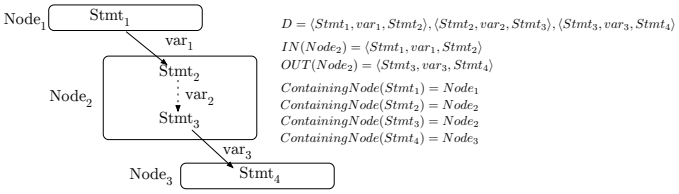

Fig. 5. Keeping track of dependencies between nodes.

1) Check Control Dependencies: Nodes in a pipeline can have both control and data dependencies. We handle control dependencies implicitly by constraining each node to have a single exit. The first stage of our analysis builds a *control flow graph* for the program and checks that each node obeys this single exit constraint. One consequence of this is that a node should not throw an exception that it does not catch. If a node violates this constraint, we inform the developer and abort the rest of the analysis.

2) Compute Data Dependencies: The second stage of the analysis computes the data dependencies between nodes and determines how to route the values of variables between nodes. For instance, in Figure 1, `Node2` consumes the values of variables `docJPEG`, `bufferedImage` and `imagePath` from `Node1`; in turn, it produces the value of variable `docTamura` for `Node3`.

To compute the data dependencies between nodes, we first build a *data dependence graph* for the method that contains the annotated nodes.[2] Let V be the set of variables and S be the set of statements in the program. A data dependence graph contains vertices that represent program statements and edges that represent data dependencies between statements. A data dependence exists between statements $s1, s2 \in S$ if $s1$ *defines* a variable $v \in V$ and $s2$ *uses* v, and there is no intervening definition of v along the execution path from $s1$ to $s2$. Let all data dependencies in the program be represented as tuples of the form $\langle s1, v, s2\rangle \in D \subseteq S \times V \times S$, where $s1$ defines the variable v that $s2$ uses.

Each node, $n \in N$ has a set of input and output data dependencies. Denote the statements in node n by the set S_n. Define the set of input dependencies for a node as $IN(n) = \{\langle s1, v, s2\rangle \in D \mid s1 \notin S_n \wedge s2 \in S_n\}$. Intuitively, $IN(n)$ denote the data dependencies coming into the current node from outside, which is why we ignore statements internal to a node (the $s1 \notin S_n$ clause). Similarly, define the set of output dependencies as $OUT(n) = \{\langle s1, v, s2\rangle \in D \mid s1 \in S_n \wedge s2 \notin S_n\}$. Intuitively, $OUT(n)$ denote the data dependencies going out of the current node, again ignoring statements internal to a node.

Finally, define the function $ContainingNode : S \rightarrow N \cup \{generator\}$ that maps statements to nodes. Recall that the original loop serves as a generator that provides the necessary initialization data that are **not** produced by any of the nodes, e.g., loop variables and instance fields. Using the functions $IN(n)$, $OUT(n)$ and $ContainingNode(s)$, we determine how to route variables between nodes. Figure 5 illustrates these functions for a simple example.

In the LIRe example, one of the elements of $IN(Node2)$ is $\langle S\#5, \texttt{docJPEG}, S\#9\rangle$, where S#X represents the statement at line X in Figure 1. $ContainingNode(S\#5) = Node1$ so we know that `docJPEG` comes from `Node1`.

If we discover a loop-carried dependency during the process of building the data dependence graph, we inform the developer and abort the rest of the analysis. A loop-carried dependency essentially *serializes* the execution since it needs to wait for the previous iteration to complete before continuing.

3) Compute side effects: This stage of the analysis computes the possible side effects to the heap for each statement. The results of this stage are used in the final stage of our analysis to determine if two nodes might have a data race. We distinguish *read* and *write* effects to objects on the heap.There is a data race between two nodes if their contained statements may access the same object and at least one of those accesses is a write.

Computing the side effects requires both call-graph and pointer analysis. Call-graph analysis attempts to approximate the calling relations between methods in a program. Pointer analysis is a compile-time analysis that attempts to determine the set of objects pointed to by a reference variable of a reference object field. The results of both analyses allow us to track read and write accesses to the heap across method calls. Call-graph and pointer analysis are highly interdependent; we rely on WALA's *on-the-fly* call-graph construction that builds the call-graph and performs pointer analysis simultaneously, yielding better precision [26].

Our call-graph analysis employs *k-object sensitivity*, a notion of context sensitivity proposed by Milanova et al. [14] and recently shown by Naik et al. [12] to be effective for detecting data races in object-oriented programs. Object-sensitivity uses the *receiver* object at an instance method invocation (the implicit `this` parameter for object-oriented programs), to distinguish different calling contexts. Static methods that lack a receiver object have an empty context ϵ. K-object sensitivity keeps track of a sequence of, at most, k, object allocation sites representing the receiver object, i.e., objects are named by the sequence $o_1, o_2, \ldots, o_k$. In general, the larger the value of k, the more precise the results, albeit at the expense of scalability. We use $k = 2$ in our analysis.

[2]We leverage WALA's use of static single assignment (SSA) form as its intermediate representation. SSA offers the same information as a data dependence graph but in a more compact representation.

Our pointer analysis is a context sensitive, flow insensitive, field sensitive and subset-based analysis [11]. Abstract objects are distinguished based on their allocation sites in a particular context, i.e., k-object sensitivity, in our case. This allows us to treat context and objects uniformly, as done in Naik et al. [12]. This uniformity plays a key role in the final stage of our analysis.

Our side effects analysis is based on an existing interprocedural mod-ref analysis [27]. We compute the effects on instance fields, static fields and array elements. We use the result from our pointer analysis to determine what objects they can point to. For arrays, our analysis is index-insensitive, i.e., it does not distinguish between individual elements of the array. More precisely, we keep track of the effects of the following *heap-accessing* statements, where x, y and i are local variables:

- **Instance Fields** $y = x.f$ (resp. $x.f = y$) that read (resp. write) instance field f of the set of objects that x can point to.
- **Static Field** $y = C.s$ (resp. $C.s = y$) that read (resp. write) to the static field s of class C.
- **Arrays** $y = x[i]$ (resp. $x[i] = y$) that read (resp. write) an element of the set of arrays that x can point to.

Define $StatementRef(s, c)$ (resp. $StatementMod(s, c)$) as the set of heap objects that statement s in context c of the enclosing method can read (resp. write). Define $MethodRef(c, m)$ (resp. $MethodMod(c, m)$) as the set of heap objects that can be read (resp. written) in context c of method m. Intuitively, $MethodRef$ and $MethodMod$ *summarize* all the heap accesses for a particular method, including its callees (as determined by the call-graph analysis) transitively. Figure 6 shows the algorithm for computing the side effects for each node in our pipeline. On line 8, we consult the call-graph to determine the possible target methods for the invocation statement in the current context. We apply the algorithm to each node in our pipeline and collect its read effects in nodeRef and write effects in nodeMod. For brevity, we write $Set_1 \leftarrow Set_1 \cup Set_2$ as $Set_1 \cup = Set_2$.

B. Check Inter-node Data Races

The final stage of the analysis makes use of information from the previous stages to check for data race between nodes. Our analysis checks for data races between *pairs* of nodes. Considering pairs of nodes at a time simplifies the analysis and also makes it easier to pinpoint and report the possible data races to the developer in an understandable way.

Let $\{(n_i, n_j) \mid n_i, n_j \in N \wedge i \neq j\}$ represent pairs of nodes in the pipeline. From the previous stage of the analysis, we computed both nodeRef and nodeMod for each node. Figure 7 presents our algorithm for computing possible data races.

Because n_i and n_j can operate in parallel, there is a possible data race if n_i reads from **or** writes to a memory location that n_2 also writes to. Line 1 of Figure 7 computes the set of objects that could have a data race. The algorithm then examines each object and prints specialize warning messages based on the type of the object and the pairs of nodes involved.

Input: callgraph, currentContext, statementsInNode
Output: nodeRef, nodeMod

```
1:  nodeRef ← ∅
2:  nodeMod ← ∅
3:  for all Statement s : statementsInNode do
4:    if isHeapAccessingStmt(s) then
5:      nodeRef ∪ = StatementRef(s, currentContext)
6:      nodeMod ∪ = StatementMod(s, currentContext)
7:    else if isMethodInvocationStmt(s) then
8:      methods ← possibleInvocations(callgraph, s, current-
        Context)
9:      for all ⟨c, m⟩ : methods do
10:       nodeRef ∪ = MethodRef(c,m)
11:       nodeMod ∪ = MethodMod(c,m)
12:     end for
13:   else
14:     {Statement has no effect on heap}
15:   end if
16: end for
```

Fig. 6. Algorithm to compute the side effects for each node.

Input: $nodeRef_{n_i}, nodeMod_{n_i}, nodeMod_{n_j}$

```
1:  conflictingHeapObjects ← (nodeRef_{n_i} ∪ nodeMod_{n_i}) ∩
    nodeMod_{n_j}
2:  for all Object o : conflictingHeapObjects do
3:    if isStaticField(o) then
4:      warningMessageForStaticField(o, n_i, n_j)
5:    else if isInstanceField(o) then
6:      warningMessageForInstanceField(o, n_i, n_j)
7:    else
8:      warningMessageForArray(o, n_i, n_j)
9:    end if
10: end for
```

Fig. 7. Basic algorithm to compute the possible data races between pairs of nodes.

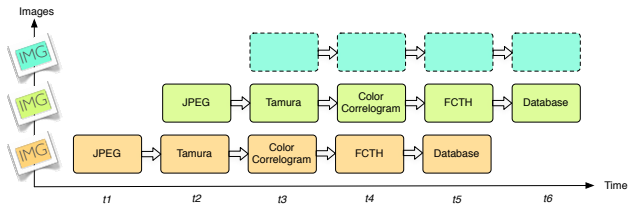

Fig. 8. The LIRe pipeline executing through time.

While the basic algorithm shown in Figure 7 works (it is sound), it is too conservative. For instance, in Figure 1, it will warn that the object pointed to by the variable `docJPEG` could be involved in a data race since it is being accessed (and at least one access is a write) in both `Node1` and `Node2`. The basic algorithm fails to take advantage of a useful property of flow-based parallelism: *ownership transfer*.

Figure 8 shows the pipeline execution of LIRe. At each time slice, *t*, at most *n* nodes can execute in parallel, where n is the

number of annotated nodes in the pipeline. Each node operates on a different image. We illustrate this by using a different color to represent which image each node is operating on at each time slice. A common pattern we have observed is that each stage receives an object, operates on it, generates new objects and passes them on to the next stage. This is a form of ownership transfer [28]. *A node transfers ownership of the object it was operating on to the next node in the pipeline.*

Input: $nodeRef_{n_i}, nodeMod_{n_i}, nodeMod_{n_j}$
1: conflictingHeapObjects $\leftarrow (nodeRef_{n_i} \cup nodeMod_{n_i}) \cap nodeMod_{n_j}$
2: **for all** Object o : conflictingHeapObjects **do**
3: **if** isStaticField(o) **then**
4: warningMessageForStaticField(o, n_i, n_j)
5: **else if** isTransferred(o, n_i, n_j) **then**
6: {No data race since object ownership is transferred}
7: **else if** isInstanceField(o) **then**
8: warningMessageForInstanceField(o, n_i, n_j)
9: **else**
10: warningMessageForArray(o, n_i, n_j)
11: **end if**
12: **end for**

Fig. 9. Algorithm that incorporates ownership transfer to compute the possible data races between pairs of nodes.

Figure 9 shows the new algorithm that incorporates ownership transfer. The main changes are on lines 5 – 6. Notice that ownership transfer only affects non-static objects. Static objects are essentially global variables and we cannot transfer their ownership since all nodes operate on the same static object. On the other hand, it is possible to transfer ownership of instance objects and arrays.

To determine if an object can be transferred between nodes, we use the results of the data dependence and pointer analysis from previous stages. To be able to transfer an object, a node must first *own* it. There are two ways that a node becomes the owner. First, a previous node could have transferred ownership to it. Second, and the most common case, is that the node created the object. For instance, from Section III-A2, we computed that the variable `docJPEG` is passed from `Node1` to `Node2`. The object pointed to by `docJPEG` was indeed created in `Node2`. Thus, it can transfer the ownership to `Node3`. Using this object as the *root*, we then calculate all other objects that could also be transferred. Figure 10 shows the procedure $isTransferred$, which determines if object o is transferred from node n_i to node n_j.

The key idea of $isTransferred$ is to take advantage of the way that objects are labeled using a k-object sensitive pointer analysis. Objects are labeled through a sequence of their allocation sites, i.e., $o_1, o_2, \ldots, o_k$. We can use this label to determine if an object is rooted at one of the objects that was transferred. If so, then by transitivity, it is also transferred. We make use of this fact on line 5 of Figure 10.

An important assumption of our algorithm is that the developer starts with a purely sequential version of the application that she wishes to parallelize. If her application already incorporates some form of parallelism (through threads), then we would need to also perform a thread escape analysis to ensure that the objects in one node are not *leaked* to other threads.

Input: dataDependenceAnalysis, pointerAnalysis, o, n_i, n_j
Output: true if the object o is transferred, false otherwise
1: var $\leftarrow \{\langle s1, v, s2\rangle \mid ContainingNode(s1) = n_i \wedge ContainingNode(s2) = n_j\}$
2: objects $\leftarrow \{pointsTo(v) \mid \langle \_, v, \_\rangle \in var\}$
3: root $\leftarrow \{r \mid r \in objects \wedge isOwner(r, n_i)\}$
4: $o_1, o_2, \ldots, o_k$ = label(o)
5: **return** true if $\exists r \in root$ that is part of the label(o)

Fig. 10. Procedure isTransferred(o, n_i, n_j)

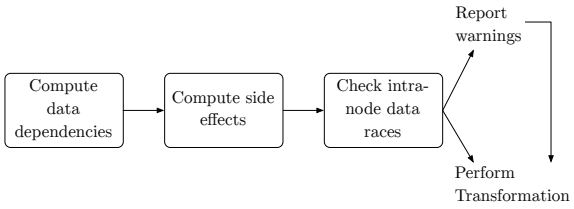

Fig. 11. Overview of the analysis workflow for the INVERT LOOP refactoring.

C. Invert Loop

The analysis for the INVERT LOOP refactoring checks if any of the extracted nodes from the EXTRACT NODE refactoring can be run in a data parallel manner. While the analysis for EXTRACT NODES checks for *inter*-node parallelism, this analysis checks for *intra*-node parallelism. Figure 11 presents an overview of the analysis. The first and second stages follow the same principles as the previous analysis; we discuss only the third stage — computing possible data races.

Input: $nodeMod_n$
1: **for all** Object o : $nodeMod_n$ **do**
2: **if** isStaticField(o) **then**
3: warningMessageForStaticField(o, n)
4: **else if** isOwner(o, n) $\vee$ isTransferred(o, n, predecessor(n)) **then**
5: {No data race on this object}
6: **else if** isInstanceField(o) **then**
7: warningMessageForInstanceField(o, n)
8: **else**
9: warningMessageForArray(o, n)
10: **end if**
11: **end for**

Fig. 12. Algorithm to determine if a node can run in a data-parallel manner.

Figure 12 shows the algorithm for computing possible data races when a node is run in a data-parallel manner. It checks all possible write effects to the heap from the selected node. Any write access to a static field will cause a data race when the node operates in parallel. However, write accesses to instance fields and arrays are safe as long as the node only writes to heap locations that it owns or has been transferred to it from a predecessor node. Intuitively, a node can run in parallel if it operates only on *fresh* objects.

If the analyses for INVERT LOOP succeeds, JFlow informs the developer of the possibility of running a node in a data-parallel manner. Recall that running a node in this manner causes its processed objects to arrive out-of-order to the next node. Thus, the developer needs to use her domain knowledge to decide if this is permissible for her application.

IV. EVALUATION

We have implemented JFlow and evaluated it on a 2.33 GHz 4-core Core 2 Quad processor with 4 GB of memory. When invoked interactively as an Eclipse plug-in, JFlow takes less than 30 seconds to perform the analysis and transformations. We think this is reasonable for an interactive tool. The bulk of this time was spent in initializing the call graph and pointer analysis.

After performing the transformations, we used Oracle's Java HotSpot 64-Bit Server JVM to measure the execution times. We set the min and max heap space for the JVM to 512M. We average the running times over 6 runs. The source code for both the serial and parallel versions of the benchmarks are available at http://vazexqi.github.io/JFlow/.

Table I shows the results. The Δ SLOC Parallel column gives an estimate of the transformation effort required to use GPars parallel library constructs. The actual parallelization effort, if done manually, is much higher since the developer has to also spend time analyzing and scrutinizing the code for data races. The performance numbers show that applications parallelize using flow-based parallelism through the constructs of a parallel library can perform reasonably well. We now discuss the necessary inspections and changes (if any) involved with parallelizing these applications using JFlow.

The first four benchmarks are taken directly from OoOJava [22], which is most similar to our work. OoOJava solves the complexity of parallelizing modern object-oriented applications through a combined static and dynamic approach with a custom runtime (see Section V). JFlow, on the other hand, solves the problem by involving developer in the parallelization process. The partitioning annotations for each program follow the task annotations from OoOJava. We use these benchmarks to validate the effectiveness of our approach in analyzing and transforming sequential code to parallel code. Note that we do not directly compare execution times as the runtime of OoOJava is sufficiently different from the approach that we are taking.

After the initial annotation for the nodes, JFlow was able to parallelize KMeans successfully without any user intervention. For MolDyn, it reported one spurious data race that involved array accesses because our analysis is index-insensitive for array accesses. For Monte Carlo, it reported spurious data races on *two* statements that access the file system. Java's File API is very complex and uses a lot of programming idioms (in particular, the Decorator design pattern for wrapping different input streams) that confuses static analysis. OoOJava does not use Java's File API; it translates Java into C and, thus, is spared from the complexity of analyzing Java's API. Finally, for RayTracer, we reported spurious data races stemming from one statement. This single statement made use of multiple objects allocated using static factory methods. Recall that for our k-object sensitive analysis, static methods are all lumped under one context, ϵ, which reduces the precision of the analysis. However, because all the spurious data races only involved a single statement and were of a particular type of problem, it was easy for a developer to inspect these by hand and proceed with the transformation.

The next three applications are what we term *real* applications; they are about an order of magnitude larger in terms of lines of code. We chose these applications because they are similar to the applications parallelized using pipeline parallelism from the PARSEC benchmark (written in C++ instead of Java) [29]. PARSEC comprises a representative set of emerging applications and we are interested to see how well JFlow can handle them. We used the parallel versions of the PARSEC benchmarks to guide our partitioning annotations.

Duke is a *data-deduplication* engine. Our benchmark processed a 700MB file looking for pairs of duplicates and links them together. Duke has a parallel version with threads. We followed the same parallelization scheme and achieve similar speedups as its threaded version, i.e., 1.49x. This shows that JFlow is able to handle how a developer might parallelize it by hand. In fact, it detected two data races that was present in the original parallelization by the author (writes to two separate counters without proper synchronization). The speedup is low because Duke is primarily I/O bound rather than CPU bound.

Jbzip2 is a bzip2 compression/decompression library. Our benchmark decompressed and compressed 1000 files; this represents a typical workflow where one would decompress a file to manipulate it and then compress it again. As with MolDyn, the main challenge here was the spurious data races reported from statements that used Java's File API. However, we could isolate these data races to particular statements that used the File API and it was easy for a developer to manually inspect by hand and proceed. Below are two lines (out of six) that were involved in compressing the contents of a file. The main source of confusion in static analysis was the *wrapping* of each stream (a key characteristic of the Decorator design pattern). While it is hard for static analysis, any developer can quickly inspect these lines and conclude that the streams all operate on different files and, thus, cannot have a data race.

```
1 OutputStream compressedStream= new BufferedOutputStream(
2  new FileOutputStream(tempFile));
3 BZip2OutputStream bzip2OStream= new BZip2OutputStream(
4  compressedStream);
```

Section II described LIRe in detail. Here we focus on

TABLE I
SPEEDUPS ON BENCHMARKS

Application	Domain	SLOC Sequential	Δ SLOC Parallel	Sequential (ms)	Parallel (ms)	Speedup
KMeans	Mining & Synthesis	504	+73	12980	4356	2.98x
Monte Carlo	Financial Trading	991	+45	14906	5294	2.82x
MolDyn	Molecular Dynamics	653	+56	23694	7029	3.37x
RayTracer	Image processing	828	+55	17704	5127	3.45x
Duke	Data warehousing	11464	+55	9502	6384	1.49x
Jbzip2	Data warehousing	3238	+38	17785	7011	2.54x
LIRe	Image processing	15476	+76	49226	15446	3.19x

the challenges to parallelization. The original version of LIRe used *reflection* to instantiate its classes based on class names. For instance, to instantiate some of its feature extractors, it used the `Class.forName(x).newInstance()` construct. This form of dynamic class creation, which is common in *dependency-injection* frameworks, poses great difficulties for static analysis [11], [30]. We manually transformed the code so that it instantiates the class directly via a `new` statement to the specific class. Like most modern applications, LIRe makes use of the highly reflective `java.util.logging` API for error reporting. Specifically it used the log4j API; we treated such calls as being thread safe and ignored data races through those calls.

Both Duke and LIRe use the Lucene search engine library for its indexing and querying. By default, JFlow does not analyze the source code for external libraries such as Lucene. We justify this decision in favor of scalability and understandability. First, external libraries are huge; moreover, many projects make use of more than one external library. Analyzing these libraries will increase the analysis time significantly. Second, even if we do analyze external libraries, data races reported that involve the internal structure will be hard to understand by a developer who is merely using it as a *black-box* library. Her efforts would be better rewarded from reading the APIs and documentation for those external libraries with regards to their thread safety. When we ignore calls to external libraries, we issue a warning to the developer so that she knows that some of the information might be missing. If she chooses, she can instruct JFlow to analyze these libraries, albeit at the expense of time and memory.

TABLE II
INSPECTIONS AND CHANGES NEEDED FROM THE DEVELOPER

Application	Inspections & changes
Kmeans	None
Monte Carlo	Inspect two statements using File API
MolDyn	Inspect one statement accessing arrays
RayTracer	Inspect one statement using objects created through static factory methods
Duke	Inspect two statements accessing Lucene API
Jbzip2	Inspect six statements using File API Convert reflective construction to use `new`
Lire	Ignore data races from logging constructs Inspect four statements accessing Lucene API

Table II summarizes the inspections and changes necessary from the developer. We believe that the effort required from the developer is minimal. If a developer were to do this entirely manually, it would have required more effort to analyze and transform. The feedback from JFlow pinpoints the potential problems and guides her parallelization efforts. We sampled seven applications from different domains and we believe that the inspections and changes required are typical of modern applications. A common pattern we observed is that most of the applications manipulate data through File APIs; future work could focus on providing more precise analysis for such operations.

V. RELATED WORK

The ideas behind flow-based parallelism are not new. The earliest influences come from the dataflow programming community. Johnston et al. provide a detailed history of the evolution of the hardware and software for dataflow programming in [31]. Similar to flow-based programming, a program in the dataflow execution model is represented by a directed graph; the nodes of the graph represent operations and the edges between nodes represent data dependencies [32]. Dataflow programming typically relies on dedicated dataflow languages. Lucid [33] and Id [34] are examples of early dataflow languages while LabView [35] and Prograph [36] are examples of current dataflow languages that are in use. Most dataflow languages tend to be restrictive, favoring functional styles without side effects [37]. These restrictions allow for easier reasoning of the system but also require that existing programs be rewritten to conform to those restrictions. Our approach with JFlow does not mandate a functional style of programming. Our approach is more practical and allows existing programs with side-effects to take advantage of flow-based parallelism as long as the side-effects are well isolated.

Morrison's original work on flow-based applications focused mostly on the architecture and design of systems using flow-based programming, i.e., decomposing a system into black boxes that communicate via message passing [1]. While he did acknowledge that flow-based programming lends itself well to parallelism, his implementation [38] focused mostly on modularity and not on parallel performance. This differs from streaming languages such as Brook [39], StreamIt [40] and StreamC/KernelC [41], where performance was the primary focus. By building upon well-designed parallel libraries, our

work attempts to bring both performance and modularity to a parallelized application.

To infer the side effects of statements, we relied on a mod-ref analysis. An alternative approach is to rely on types and effects systems [42]–[44]. While types and effects systems can be more precise, an inherent difficulty is that they require extensive annotations from the developer. Work has been done to alleviate this by inferring some of these annotations [45]–[47]. Another limitation of such type systems is that they usually require custom language syntax. This makes them challenging to adopt into existing programs written in legacy programming languages. In this regard, approaches based on *pluggable type systems* are attractive because they allow a type system to be seamlessly added to an existing programming language [48].

We rely on static analysis to determine if an application can be parallelized safely. As demonstrated in Section IV, static analysis tends to be conservative. Realizing the limitations of static analysis, some researchers have proposed a dynamic approach to program analysis through profiling at runtime. These approaches rely on the developer to provide a representative test program for the data dependence analyses to work. A faulty test program could easily jeopardize the entire analysis and transformation process.

Though unsound, dynamic approaches can be quite effective. Thies et al. [21] first proposed an annotation-based method for automatically detecting and parallelizing pipeline parallelism in C programs. Rul et al. [49] and Tournavitis et al. [50] improved on their work and automatically detect and parallelize pipeline parallelism in applications without any annotations. Unfortunately, this approach is not always ideal for every workflow. Running, collecting and analyzing the data is slow and consumes a lot of memory and storage. Work by Kim et al. attempt to alleviate this overhead by parallelizing and compressing the data collection process [51].

Static and dynamic analysis need not be mutually exclusive. The OoOJava project combines both [22]. It uses *disjoint reachability* analysis to statically analyze the code. When the analysis cannot safely prove that threads operate on disjoint objects, it inserts a run-time check. During run time, it compares the addresses of both objects to determine if they are indeed disjoint. Such an approach requires the use of a specialized runtime and might not be feasible for all scenarios. In its current implementation, OoOJava sidesteps some of the complexities of analyzing the internal Java libraries (e.g. Java File API) by translating programs into C and using simpler library calls.

The idea of interactive tools for parallelization has been explored in various forms. Two notable tools are SUIF Explorer [52] and Parascope Editor [53], which focus on transformations for loop parallelism for scientific applications written in Fortran. Both tools allow the developer to examine the outcomes of the analysis of the compiler in a rich environment that includes various visualizations for program dependencies. Both tools utilized deep interprocedural analyses, which are inherently conservative, and sought developer input to help refine the outcomes of the analyses.

More recently, Dig et al. demonstrated that a transformation-based approach for introducing concurrency and parallelism into sequential code via library constructs is practical. Their initial work focused on providing support for three kinds of transformations: (i) convert `int` to `AtomicInteger`, (ii) convert `HashMap` to `ConcurrentHashMap` and (iii) convert recursion to `ForkJoinTask` [54]. Dig et al. then created RELOOPER, a tool for for converting array operations to use data parallelism through the new `ParallelArray` construct in Java [55]. Kjolstad et al. continued work in this area by introducing transformations to convert mutable classes into immutable classes, i.e., value objects [56].

VI. Conclusions and Future Work

Modern applications are complex. They tend to use many different programming idioms and many library APIs. Relying solely on fully automated parallelizing compilers is likely to fail and yield unsatisfactory results. On the other hand, the interactive approach taken by JFlow is both effective and practical. By leveraging a fast analysis, it tightens the feedback loop, allowing developers to invoke the tool, act on the feedback and repeat the process until they have successfully parallelized their application. While static analysis is constantly improving to account for more programming idioms, it can never handle *all* of them — nor should it. Incorporating the developer as part of a tool's workflow complements static analysis and is a practical approach that yields great benefits.

This paper adopts the interactive approach for refactoring sequential programs to use pipeline parallelism, the most common form of flow-based parallelism that we have observed. However, this only scratches the surface of flow-based parallelism. Many more styles of flow-based parallelism exist and it would be helpful to equip developers with tools to handle them. Future work in this area falls into two key themes: *decomposition* and *re-composition*.

Decomposition focuses on helping the developer partitioning the original sequential program into nodes of a computation flow graph. Currently we rely on the developer to annotate the nodes in a pipeline. This works well for a pipeline with its linear structure. However, many more complex structures exist and work remains to determine how to handle them. Re-composition, on the other hand, focuses on helping the developer transform the partitioned program to run in parallel. This involves composing the most appropriate parallel library constructs to use and tuning them not only for performance but also for extensibility.

Acknowledgment

The authors would like to thank Stas Negara, Mohsen Vakilian, Gopal Santhanaraman and Fredrik Kjolstad for their thoughtful comments and feedback on the ideas and the implementation. Nicholas Chen was supported by US Department of Energy Grant No. DOE DE-FG02-06ER25752 and the Ray Ozzie Fellowship.

REFERENCES

[1] J. P. Morrison, *Flow-Based Programming: A New Approach to Application Development*, 2nd ed. CreateSpace, 2010.

[2] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[3] E.-G. Kim and M. Snir, "Resources on Parallel Patterns," http://www.cs.uiuc.edu/homes/snir/PPP/, 2011.

[4] "GPars (Groovy Parallel Systems)," http://gpars.codehaus.org/.

[5] Intel Corporation, "Intel Threading Building Blocks," http://www.threadingbuildingblocks.org/.

[6] "TPL Dataflow," http://msdn.microsoft.com/en-us/devlabs/gg585582.

[7] E. Reed, N. Chen, and R. Johnson, "Expressing Pipeline Parallelism Using TBB Constructs: A Case Study on What Works and What Doesn't," in *TMC*, 2011.

[8] M. Kim, H. Kim, and C.-K. Luk, "Prospector: A Dynamic Data-Dependence Profiler To Help Parallel Programming," in *HotPar*, 2010.

[9] A. J. Dorta, C. Rodriguez, F. d. Sande, and A. Gonzalez-Escribano, "The OpenMP Source Code Repository," in *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2005, pp. 244–250.

[10] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[11] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, *Alias Analysis for Object-Oriented Programs*. Springer Berlin Heidelberg, 2013, vol. 7850, pp. 196–232.

[12] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," in *PLDI '06*.

[13] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '99. IBM Press, 1999, pp. 13–.

[14] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 1–41, January 2005.

[15] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1981, ch. 7, pp. 189–234.

[16] T. B. Sheridan, "Function allocation: algorithm, alchemy or apostasy?" *Int. J. Hum.-Comput. Stud.*, vol. 52, pp. 203–216, February 2000.

[17] M. Lux and S. A. Chatzichristofis, "Lire: Lucene Image retrieval - An Extensible Java CBIR library," in *MM '08*.

[18] H. Vandierendonck, S. Rul, and K. De Bosschere, "The Paralax Infrastructure: Automatic Parallelization with a Helping Hand," in *PACT '10*.

[19] V. Sarkar, "Automatic partitioning of a program dependence graph into parallel tasks," *IBM J. Res. Dev.*, vol. 35, no. 5-6, pp. 779–804, Sep. 1991.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.

[21] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs," in *MICRO '07*.

[22] J. C. Jenista, Y. h. Eom, and B. C. Demsky, "OoOJava: Software Out-of-order Execution," in *PPoPP '11*.

[23] K. Knobe, "Ease of use with concurrent collections (CnC)," in *HotPar '09*, Berkeley, CA, USA, 2009.

[24] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-Directed Pipeline Parallelism," in *PACT '10*, 2010.

[25] "T.J. Watson Libraries for Analysis (WALA)," http://wala.sf.net.

[26] D. Grove and C. Chambers, "A Framework for Call Graph Construction Algorithms," *ACM Trans. Program. Lang. Syst.*, vol. 23, pp. 685–746, November 2001.

[27] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher, "A Schema for Interprocedural Modification Side-Effect Analysis With Pointer Aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 2, pp. 105–186, Mar. 2001.

[28] S. Negara, R. K. Karmani, and G. Agha, "Inferring Ownership Transfer for Efficient Message Passing," in *PPoPP '11*. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941566

[29] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[30] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders," in *ICSE '11*.

[31] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in Dataflow Programming Languages," *ACM Comput. Surv.*, vol. 36, pp. 1–34, March 2004.

[32] A. Davis and R. Keller, "Data Flow Program Graphs," *Computer*, vol. 15, no. 2, pp. 26 – 41, feb 1982.

[33] W. W. Wadge and E. A. Ashcroft, *LUCID, The Dataflow Programming Language*. San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[34] Arvind, K. P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," University of California Irvine, Tech. Rep., 1978.

[35] National Instruments, "What is NI Labview?" http://www.ni.com/labview/whatis/.

[36] S. Matwin and T. Pietrzykowski, "PROGRAPH: A preliminary report," *Computer Languages*, vol. 10, no. 2, pp. 91 – 126, 1985.

[37] W. Ackerman, "Data Flow Languages," *Computer*, vol. 15, no. 2, pp. 15 – 25, feb 1982.

[38] J. P. Morrison, "Java Flow-Based Programming," http://sourceforge.net/projects/flow-based-pgmg.

[39] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," in *SIGGRAPH '04*.

[40] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *CC '02*.

[41] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *Computer*, vol. 36, pp. 54–62, August 2003.

[42] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A Type and Effect System for Deterministic Parallel Java," in *OOPSLA '09*.

[43] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou, "Using Data Groups to Specify and Check Side Effects," in *PLDI '02*.

[44] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems," in *POPL '88*, New York, NY, USA.

[45] N. Kellenberger, "Static Universe Type Inference," Master's thesis, ETH Zurich, 2005.

[46] M. Niklaus, "Static universe type inference using a SAT-solver," Master's thesis, ETH Zurich, 2006.

[47] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson, "Inferring Method Effect Summaries for Nested Heap Regions," in *ASE '09*, Washington, DC, USA.

[48] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller, "Building and Using Pluggable Type-Checkers," in *ICSE '11*, New York, NY, USA.

[49] S. Rul, H. Vandierendonck, and K. De Bosschere, "A Profile-Based Tool for Finding Pipeline Parallelism in Sequential Programs," *Parallel Comput.*, vol. 36, pp. 531–551, September 2010.

[50] G. Tournavitis and B. Franke, "Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism using Profiling Information," in *PACT '10*, New York, NY, USA.

[51] M. Kim, H. Kim, and C.-K. Luk, "SD3: A Scalable Approach to Dynamic Data-Dependence Profiling," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 535–546.

[52] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam, "SUIF Explorer: An Interactive and Interprocedural Parallelizer," in *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '99. New York, NY, USA: ACM, 1999, pp. 37–48.

[53] K. Kennedy, K. S. McKinley, and C. W. Tseng, "Interactive Parallel Programming using the ParaScope Editor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 329–341, July 1991.

[54] D. Dig, J. Marrero, and M. D. Ernst, "Concurrencer: A Tool for Retrofitting Concurrency into Sequential Java Applications via Concurrent Libraries," in *ICSE '09 (Companion)*.

[55] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "Relooper: Refactoring for Loop Parallelism in Java," in *OOPSLA '09 (Companion)*, New York, NY, USA.

[56] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Transformation for Class Immutability," in *ICSE '11*, New York, NY, USA.